

[ASP.NET AJAX Sample Website](#)

Threading--References

[CLR via C#, by Jeffrey Richter \(Microsoft Press\)](#)

<http://msdn.microsoft.com/msdnmag/issues/03/01/NET/>

<http://msdn.microsoft.com/msdnmag/issues/06/06/ConcurrentAffairs/>

<http://blogs.iis.net/tobintitus/archive/2006/10/05/Unsafe-thread-safety.aspx>

<http://hacked.com/archive/2006/08/08/ThreadingNeverLockThisRedux.aspx>

### Multithreading

- Threads are the basic unit to which the operating system allocates processor time.
- More than one thread can be executing code inside the same process (application).
- On a single-processor machine, the operating system is switching rapidly between the threads, given the appearance of simultaneous execution.
- If multiple threads are accessing the same resources, synchronization may be necessary.

### When to Use Multithreading

With threads, you can:

- Maintain a responsive user interface while background tasks are executing
- Distinguish tasks of varying priority
- Perform simultaneous communications tasks: to a database, to a web service, etc.
- Perform operations that consume a large amount of time without stopping the rest of the application

### Disadvantages of Multiple Threads

Multiple threads:

- Prone to errors
- Difficult to debug
- Lots of threads consume lots of processor time
- Synchronization is necessary to coordinate resource sharing
- Execution time is generally not reduced on a single-processor machine

### Starting a Thread

```
using System.Threading;

Thread thread = new Thread(MyStartFunc);
thread.Start();
...
```

Function that will execute when the thread starts

```
void MyStartFunc()
{
    ...
}
```

No arguments, no return value

Wrapped in a ThreadStart delegate by inference

### Starting a Thread

```
using System.Threading;
class MyClass{
private int x;
Thread thread = new Thread(MyStartFunc);
thread.Start();
...

void MyStartFunc()
{
    x = ...
}
```

### Starting a Thread

```
using System.Threading;

Z myZ = new Z(4);
Thread thread = new Thread(myZ.MyStartFunc);
thread.Start();
...

class Z
{
    private int initData;

    public Z (int someInitData)
    {
        initData = someInitData;
    }

    public void MyStartFunc()
    {
        //can access initData here
    }
}
```

### Starting a Thread

```
using System.Threading;
Z myZ = new Z(4);
Thread thread = new Thread(myZ.MyStartFunc);
thread.Start();
...
```

Pass in the initialization data when instantiating the class

```
class Z
{
    private int initData;
    public Z (int someInitData)
    {
        initData = someInitData;
    }
    void MyStartFunc()
    {
        //can access initData here
    }
}
```

Thread method is in a separate class and can access its fields, which can be set in the constructor

### Thread Properties

		Get	Set	Static
CurrentThread	returns a reference to the calling thread	+		+
IsAlive	true if started and not terminated		+	
IsBackground	true if a background thread		+	+
Name	human-readable name (default = null)		+	+
Priority	priority (default = ThreadPriority.Normal)		+	+
ThreadState	thread state (Running, Aborted, ...)		+	

### Suspending and Resuming a Thread

```
thread.Suspend()
```

- temporarily suspends a running thread
- there is no "suspend count"
- a thread can suspend itself, but in that case another thread will have to restart it

```
thread.Resume()
```

- restarts a suspended thread

### Terminating a Thread

```
thread.Abort()
```

- throws ThreadAbortException to begin process of terminating thread using exceptions allows threads to clean up before termination
- a thread can Abort itself

```
thread.Join()
thread.Join(int)
thread.Join(TimeSpan)
```

- blocks the caller until the thread terminates or a timeout occurs

### Terminating a Thread

```
myThread.Abort(); //ask thread to terminate
myThread.Join(); //wait until it does
```

### Terminating a Thread

```
myThread.Abort(); //ask thread to terminate
myThread.Join(); //wait until it does
```

```
void MyThreadFunction()
{
    ...
    try
    {
        sqlConn.Open();
        ...
    }
    finally
    {
        sqlConn.Close();
    }
}
```

### The Classic Thread Situation

- A long computation needs to take place
- The UI must remain responsive
- UI controls have "thread affinity"
  - message queue owned by control's creating thread
- Control interaction from a thread other than the creator may not work properly

### Thread Safety

Thread safety means that the fields of an object or class always maintain a valid state, as observed by other objects and classes, even when used concurrently by multiple threads.

To achieve thread safety, we need to synchronize the activities of the threads using the same fields.

### Thread Synchronization

#### Monitors

Monitors allow us to obtain a lock on a particular object and use that lock to restrict access to a critical section of code.

While a thread owns a lock for an object, no other thread can acquire that lock.

`Monitor.Enter(object)` claims the lock, but blocks if another thread already owns it.

`Monitor.Exit(object)` releases the lock.

### Thread Synchronization

#### Monitors

```
void SomeMethod()
{
    ...
    Monitor.Enter(obj);

    critical section

    Monitor.Exit(obj);
}
```

### Thread Synchronization

#### Monitors

```
void SomeMethod()
{
    ...
    Monitor.Enter(obj);
    try
    {
        critical section
    }
    finally
    {
        Monitor.Exit(obj);
    }
}
```

### Thread Synchronization

#### lock

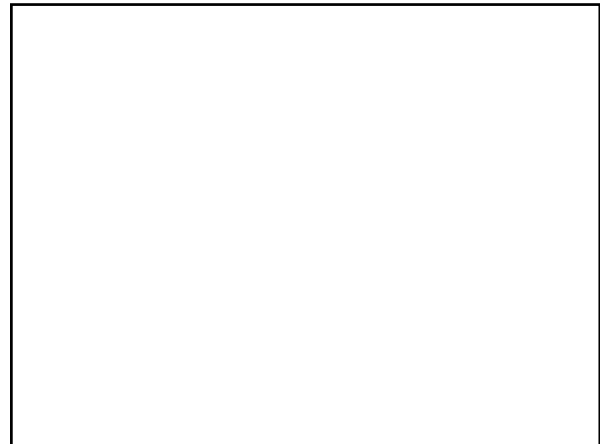
```
void SomeMethod()
{
    ...
    lock(obj)
    {
        critical section
    }
}
```

### Thread Synchronization

lock

```

void SomeMethod()          void SomeOtherMethod()
{
    ...                    {
    lock(obj)              lock(obj)
    {
        critical section   critical section
    }
}
    
```



```

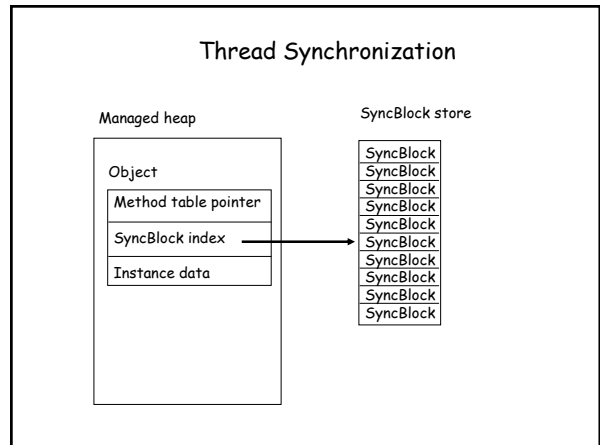
internal sealed class TransactionWithLockObject {
    // Allocate a 'private' object used for locking
    private Object m_lock = new Object();

    // Field indicating the time of the last transaction performed
    private DateTime timeOfLastTransaction;

    public void PerformTransaction() {
        lock (m_lock) { // Enter the private field object's lock
            // Perform the transaction...

            // Record time of the most recent transaction
            timeOfLastTransaction = DateTime.Now;
        } // Exit the private field object's lock
    }

    // Read-only property returning the time of the last transaction
    public DateTime LastTransaction {
        get {
            lock (m_lock) { // Enter the private field object's lock
                return timeOfLastTransaction; // Return the date/time
            } // Exit the private field object's lock
        }
    }
}
    
```



- ### Thread Synchronization
- #### Reader/Writer locks
- Permit multiple threads to read concurrently
  - Prevent overlapping reads and writes
  - Prevent overlapping writes
  - Reader function uses `AcquireReaderLock`  
`ReleaseReaderLock`
  - Writer function uses `AcquireWriterLock`  
`ReleaseWriterLock`

- ### Thread Synchronization
- #### Reader/Writer locks
- Richter recommends not using Reader/Writer locks
- Performance is slow
  - Readers are favored over writers
  - Doesn't allow one thread to enter the lock and another to complete the operation and exit the lock

### Other things to know about threading

- You can synchronize access to entire methods

```
[MethodImpl (MethodImplOptions.Synchronized)]
byte[] TransformData(byte[] buffer)
{
    ...
}
```

Only one thread at a time can enter the method. Similar to the classical notion of a "critical section".

The trouble is, this is the same thing as `lock(this)`, so it is not a good idea.

### Double-Check Locking for singleton objects

```
public static class Program {
    public static void Main() {
        Singleton s = Singleton.Value;
    }
}

public sealed class Singleton {
    private static Object s_lock = new Object();

    private static Singleton s_value;

    private Singleton() {}

    public static Singleton Value {
        get {
            if (s_value == null) {
                lock (s_lock) {
                    if (s_value == null) {
                        s_value = new Singleton();
                    }
                }
            }
            return s_value;
        }
    }
}
```

```
public sealed class Singleton
{
    private static Singleton s_value = new Singleton();

    private Singleton() { }

    public static Singleton Value
    {
        get
        {
            return s_value;
        }
    }
}
```

### Using the thread pool avoids the overhead of creating dedicated threads.

- All threads in the managed pool are background threads
- There is no way to cancel a work item after it has been queued

### Reasons for using a dedicated thread

- The thread needs to run at a special priority
- The thread needs to be a foreground thread
- The thread may need to be aborted